oRis : un environnement de simulation interactive multi-agents

Fabrice Harrouet — Jacques Tisseau — Patrick Reignier
Pierre Chevaillier

Laboratoire d'Informatique Industrielle
École Nationale d'Ingénieurs de Brest
BP 30815 — 29608 Brest Cedex
{harrouet, tisseau, reignier, chevaillier}@enib.fr — www.enib.fr/LI2

RÉSUMÉ. *oRis est un environnement de simulation interactive : c'est un langage de programmation par objets concurrents et un environnement d'exécution. Ces caractéristiques en font une plate-forme généraliste pour l'implémentation de systèmes multi-agents (SMA), plus particulièrement dédiée à la simulation. C'est un langage dynamiquement interprété, à granularité instance qui permet d'intervenir en cours de simulation pour observer le SMA, interagir avec les agents ou sur l'environnement et les modifier en ligne. En oRis, un SMA est composé d'agents (à la base des objets actifs) dont l'environnement est constitué d'objets, éventuellement situés dans l'espace (2D ou 3D) et le temps. oRis offre une solution homogène pour les interactions, qu'elles soient implémentées par appel de méthode, lien réflexes, ou envoi de message (en point-à-point ou par diffusion, avec traitement synchrone ou asynchrone). oRis implémente différents modes de gestion de flots d'activité et l'ordonnanceur garantit un partage équitable du temps entre ces flots. oRis est stable et opérationnel. Il a été utilisé dans de nombreux projets et constitue le cœur de la plate-forme de réalité virtuelle ARéVi.*

ABSTRACT. *oRis is a toolkit for interactive simulation : it is both an object-based concurrent programming language and an execution environment. Its features make oRis a generic platform for multiagent systems (MAS) implementation. It is a dynamically interpreted language, instance-grained which allows the user, during the execution, to observe the MAS, to interact with the agents or the environment et to modify them in line. With oRis, a MAS is compouned of agents (basically active objects) in an environment containing objets, eventually situated in space (2D or 3D) and time. oRis offers an homogenous solution for interactions, implemented as method invocation or callback or message passing (point-to-point or broadcast, synchronous or asynchronous processing). oRis offers different ways to manage the execution flows and the scheduler guarantees the equity of the time-sharing. oRis is stable and efficient. It has been used in many projects and is integrated in the ARéVi virtual reality platform.*

MOTS-CLÉS : *langage de programmation, programmation par objets concurrents, plate-forme, réalité virtuelle, systèmes multi-agents, simulation interactive.*

KEY WORDS: *programming language, object-based concurrent programming, toolkit, virtual reality, multiagent systems, interactive simulation.*

1. Introduction

L'environnement de simulation multi-agents que nous proposons dans cet article s'intègre dans nos travaux sur la réalité virtuelle. Un univers virtuel est un ensemble de modèles numériques autonomes en interaction, auquel les humains participent en tant qu'*avatars*. La création de ces univers repose sur un principe d'autonomie selon lequel les modèles numériques sont dotés de capteurs virtuels pour percevoir les autres modèles, possèdent des actionneurs pour agir sur les autres, disposent de moyens pour communiquer entre eux, et maîtrisent leurs coordinations perceptions-actions à travers un module de décision. Un avatar est alors un modèle numérique dont les capacités de décision sont déléguées à l'opérateur qu'il représente. Les modèles numériques, situés dans l'espace et dans le temps, évoluent ainsi de manière autonome au sein de l'univers virtuel, dont l'évolution d'ensemble est le résultat de leur évolution conjointe. L'opérateur est à la fois spectateur, acteur et créateur de l'univers virtuel auquel il appartient. Il est en relation avec son avatar par l'intermédiaire de périphériques sensori-moteurs variés et est immergé dans un environnement multisensoriel.

Le statut de l'utilisateur en réalité virtuelle est donc différent de celui qu'il peut avoir en simulation scientifique ou avec des simulateurs d'entraînement. En simulation scientifique, l'utilisateur intervient avant pour fixer les paramètres du modèle, et après pour interpréter les résultats du calcul. Il peut observer l'évolution des calculs, mais il reste cependant *esclave* du modèle. Les systèmes de simulation scientifique sont des systèmes centrés-modèle. À l'inverse, les simulateurs d'entraînement sont essentiellement centrés-utilisateur pour donner à celui-ci tous les moyens nécessaires au contrôle et au pilotage du modèle : le modèle doit demeurer *esclave* de l'utilisateur. En introduisant la notion d'avatar, la réalité virtuelle place l'utilisateur au même niveau conceptuel que le modèle. La relation *maître-esclave* est ainsi supprimée au profit d'une plus grande autonomie des modèles, et par voie de conséquence, d'une plus grande autonomie de l'utilisateur.

Cette approche de la réalité virtuelle rend opérationnelle la simulation multi-agents lorsque celle-ci nécessite que l'opérateur humain fasse partie intégrante du modèle, ce que nous appelons ici la *simulation interactive*. Dans [TIS 01], Tisseau développe une réflexion épistémologique sur ce sujet qu'il serait hors de propos d'exposer ici. L'univers multi-modèles est un système multi-agents dans lequel des opérateurs humains sont immergés au sein d'instances de modèles (des agents) s'exécutant de manière autonome et concurrente. Conformément à l'approche « voyelles » [DEM 95], ces **A**gents sont bien sûr en **I**nteraction dans un **E**nvironnement (ici numérique), suivant une **O**rganisation sociale explicite ou émergente. Cette dernière facette n'étant pas toujours explicite dans les modèles de **S**ystèmes **M**ulti-**A**gents (SMA), elles ne font pas partie des concepts de base de notre outil. Une des originalités de notre travail tient dans le rôle prépondérant joué par l'utilisateur (via son avatar).

Pour jouer librement ses trois rôles (spectateur, acteur, créateur), tout en respectant l'autonomie des agents avec lesquels il cohabite, l'opérateur doit disposer d'un *langage* pour agir sur les autres modèles, les modifier et éventuellement créer de nou-

velles classes de modèles et les instancier. Il doit donc disposer en cours d'exécution de la simulation du même pouvoir d'expression que le créateur du modèle initial. Le langage doit donc être *dynamiquement interprété* : du nouveau code doit pouvoir être introduit (et donc interprété) en cours d'exécution. L'utilisateur peut être amené à n'interagir ou à ne modifier qu'une instance d'un modèle. Le langage doit donc avoir une *granularité instance*, ce qui suppose de disposer d'un service de nommage et surtout de la possibilité, au niveau syntaxique, de distinguer le code du contexte global, des classes et des instances.

La simulation multi-agents impose d'exécuter en parallèle plusieurs modèles. Il faut donc s'assurer que le procédé d'activation de ces entités autonomes n'induit pas un biais qui risquerait d'amener le système dans un état global dont la plate-forme d'exécution serait responsable : il est impératif que seuls les procédés algorithmiques décrits dans les comportements des agents expliquent l'état global du modèle.

Il apparaît donc que la simulation multi-agents interactive impose de disposer d'un langage d'implémentation qui intègre bien sûr le paradigme de la programmation par objets actifs concurrents (l'intérêt de ce paradigme pour les SMA n'est pas une idée nouvelle [GAS 92]) mais qui offre aussi des propriétés supplémentaires : un langage dynamiquement interprété, une *granularité instance* et un mécanisme *neutre* de simulation du parallélisme. Aucun environnement de simulation n'offrant l'ensemble de ces propriétés, ceci nous donc a conduit, après les avoir clairement définies, à développer la plate-forme de simulation interactive multi-agents *oRis*. En effet, toutes les plates-formes n'offrant que des implémentation d'objets actifs ou d'agents reposant sur les *threads* système ou *Java* ne garantissent pas l'équité de l'ordonnancement des agents. Les caractéristiques des langages objets tels que *C++* et *Java* ne permettent de modifier en ligne une application qu'à partir de « points d'entrée » prédéfinis à la compilation, et ce tant au niveau classe qu'au niveau instance. Quant à *Smalltalk'80*, il répond au premier point mais pas au second.

oRis est un *langage d'implémentation* qui permet d'écrire des programmes à base d'objets et d'**A**gents (section 2), situés dans un **E**nvironnement spatio-temporel (section 3), en **I**nteraction (section 4), soumis aux actions de l'**U**tilisateur (section 5). C'est également un *environnement de simulation* qui permet le contrôle de l'ordonnancement des agents (section 6) et la modélisation interactive grâce à la dynamicité du langage (section 7). *oRis* est stable et opérationnel et a déjà donné lieu à nombre d'applications (section 8).

2. *oRis* : un langage d'implémentation

2.1. *Positionnement*

Le développement du génie logiciel a conduit à la définition de modèles, de méthodes et de méthodologies qui sont rendus opérationnels par les nombreux outils (logiciel, référentiels documentaires) qui les mettent en œuvre. Parler « d'environnement de développement de logiciel » serait par trop vague : aucun référentiel et *a fortiori*

aucun outil ne couvre l'ensemble des besoins. Dans le domaine plus restreint du développement de systèmes multi-agents (SMA), on retrouve cette même diversité avec des méthodes (voir [IGL 98] pour une revue sur ce thème), des modèles (par exemple un modèle de coordination d'actions comme les réseaux de contrats), des langages (citons par exemple *MetateM* [FIS 94], *ConGolog* [DEG 00]), des générateurs d'applications comme *ZEUS* [NWA 99], des bibliothèques de composants comme *JATLite*[1], des simulateurs de SMA comme *SMAS*[2], des « boîtes à outils » qui sont généralement des paquetages de classes offrant des services fondamentaux (cycle de vie des agents, communication, transport de messages...) et des classes implémentant certaines composantes d'un SMA ([BOI] décrit un ensemble de plates-formes développées par la communauté scientifique francophone ; [PAR 99] référence d'autres outils de ce type).

Pour Shoham, un « système de programmation orienté agents » est constitué des trois éléments fondamentaux suivants [SHO 93].

1. Un langage formel de description des « états mentaux » des agents qui sont, pour Shoham, des *modalités* telles que les *croyances* ou les *engagements*. Ceci conduit à définir une logique modale tel que le fait l'auteur dans son langage *Agent-0* ou Fisher dans *MetateM* [FIS 94] ; les états internes d'un agent réactif peuvent quant à eux être décrits comme un **S**ystème à **É**vénements **D**iscrets (SED) tel qu'un réseau de Petri comme cela a été fait dans [CHE 99a].

2. Un langage de programmation des comportements des agents qui doit respecter la sémantique des états mentaux ; ainsi, *Agent-0* définit-il une interprétation de la mise à jour des états mentaux – qui repose sur la *théorie des actes de langage* – et de l'exécution des engagements. Il serait envisageable dans ce cadre d'utiliser *KQML* et, si des contraintes d'hétérogénéité ou de standardisation l'imposaient, un ACL respectant les spécifications de la *FIPA*[3]. Dans le cas d'un SED, il conviendrait de définir la sémantique de la machine à états (e.g. une interprétation du réseau de Petri).

3. Un « agentifieur » qui permet de convertir un composant « neutre » en un agent programmable ; Shoham avoue avoir peu d'idées sur ce point. Reformulons cette exigence en disant qu'il est nécessaire de disposer d'un langage permettant de rendre les agents « opérationnels » i.e. en faire des composants implémentables et exécutables. Nous préférons considérer ce troisième élément comme un *langage d'implémentation*.

Les deux premiers éléments – qui sont intimement liés – constituent des domaines de recherche ouverts et reposent sur des choix forts quant aux types de SMA que l'on souhaite développer. Intégrer ces éléments dans une plate-forme à vocation généraliste conduit inévitablement à deux écueils : l'instabilité de la plate-forme, tant le domaine est encore ouvert, et le « diktat » d'un modèle nécessairement pas adapté à toutes les situations.

Concernant ce troisième élément, plusieurs choix sont possibles ; ils sont conditionnés par le type d'application envisagé et par les caractéristiques technologiques

1. Java Agent Template Lite, `java.stanford.edu/java_agent/html`
2. `www.hds.utc.fr/~barthes/SMAS`
3. Foundation for Intelligent Physical Agents : `www.fipa.org`

des environnements d'exécution des agents, ce qui conduit à la typologie (non exhaustive) suivante qui s'inspire de celle donnée par [NWA 96].

— Résolution de problèmes :

– agents embarqués (e.g. agents physiques (contrôle de processus), agents d'interface) : il y a de fortes contraintes d'efficacité d'exécution des programmes et de dépendance vis à vis de la technologie du système cible (nombre d'applications sont écrites en *C* ou en *C++*) ;

– agents mobiles : il est préférable de disposer d'un langage interprétable sur une large communauté de systèmes d'exploitation ; l'utilisation de langages de script tels que *Perl* peut être envisagée ;

– agents rationnels : la programmation en logique avec un langage comme *Prolog* peut parfaitement répondre à ce type de besoin.

— Simulation : Il est nécessaire de disposer d'un environnement permettant d'exécuter en parallèle le comportement des différents agents et offrant une large gamme de composants pour l'interface avec les utilisateurs. Comme le montre la lecture de [BOI], dans de nombreux cas, le choix s'est porté sur des langages s'exécutant sur une machine virtuelle autorisant la programmation parallèle : historiquement *Smalltalk'80* et, de plus en plus, *Java* (l'évolution de la plate-forme *DIMA* est révélatrice de cette tendance [GUE 99]).

Nous avons bien conscience que cette classification, comme beaucoup d'autres d'ailleurs, a quelque chose d'arbitraire et que tel ou tel langage peut s'appliquer à différents domaines (« on peut tout faire en assembleur ! ») et qu'il existe plusieurs solutions à un même problème. Elle n'a de prétention que d'éclairer le lecteur sur le positionnement d'*oRis* en tant que plate-forme multi-agents.

oRis[4] a été conçu pour répondre à la fois au besoin d'un langage d'implémentation de systèmes multi-agents et aux exigences de la simulation interactive. Pour différentes raisons expliquées dans cet article, il nous est apparu que le paradigme objet offrait le meilleur compromis dans ce cadre, mais que l'utilisation directe des langages à objets « classiques » tels que *C++* ou *Java* n'offrait pas une solution opérationnelle correspondant à nos besoins.

2.2. *Principales caractéristiques*

oRis est un langage orienté objets à typage fort et interprété. Il a d'ailleurs de nombreuses similitudes avec les langages *C++* et *Java* ce qui facilite son apprentissage. Tout comme ces langages généralistes, *oRis* permet d'aborder des thèmes applicatifs variés ; si l'on intègre des composants développés avec ces langages, l'architecture logique des applications reste homogène, ce qui facilite la réutilisabilité de composants tiers et l'extensibilité de la plate-forme *oRis*. La figure 1 propose, à titre d'illustration, un programme minimal, mais néanmoins complet, définissant une classe et lançant

4. Le suffixe latin *oris* signifie « celui qui agit ».
Exemple : *cantatio* → chant, *cantoris* → chanteur (« celui qui chante »).

quelques traitements (blocs `start`). Nous pouvons remarquer que la classe décrit des objets actifs dont les comportements s'exécutent en parallèle d'autres traitements initiés dans d'autres contextes locaux.

oRis dispose d'un *ramasse-miettes*. Il est possible de choisir quelles instances sont sujettes à une destruction automatique (cette décision est révocable dynamiquement), la destruction explicite d'une instance par l'opérateur `delete` étant toujours possible. Il existe bien entendu un mécanisme permettant de savoir si une référence est toujours valide.

```
class MyClass                                        // definir la classe MyClass
{
string _txt;                                         // un attribut
void new(string txt) { _txt=txt; }                   // constructeur
void delete(void)    {}                              // destructeur
void main(void)      { println("I say: ",_txt); } // comportement
}

start                                                // initier un traitement
{                                                    //   dans l'application
println("---- block start ----");
MyClass i=new MyClass("Hello");                      // creer des objets actifs
MyClass j=new MyClass("World");
println("---- block end ----");
}

start                                                // initier un autre
{                                                    //   traitement
for(int i=0;i<100;i++)
  {
  println("doing something else !");
  yield();
  }
}
```

```
doing something else !
---- block start ----
---- block end ----
I say: World
doing something else !
I say: Hello
I say: Hello
I say: World
doing something else !
I say: Hello
I say: World
doing something else !
...
```

Figure 1. *Exécution d'un programme simpliste en oRis*

Nous disposons d'une interface avec le langage *C/C++* selon un procédé de bibliothèques chargeables à la demande. Non seulement *oRis* permet d'associer des fonctions ou des méthodes à une implémentation en *C/C++*, mais il permet une utilisation plus en avant de l'approche *objets* en matérialisant directement une instance du langage interprétés par une instance décrite en *C++*. Bien qu'*oRis* soit implémenté en *C++* il est possible d'intégrer des travaux réalisés dans d'autres langages. Un *paquetage* assure l'interfaçage du code *oRis* avec du code *SWI-Prolog* : une fonction permet d'invoquer l'interpréteur *Prolog* et, en *Prolog*, il est possible d'invoquer une méthode *oRis*. *oRis* a également été interfacé avec *Java* de telle sorte que n'importe quelle classe *Java*, qu'elle fasse partie des classes standards ou qu'elle soit l'objet d'un travail personnel, peut être directement utilisée depuis *oRis*. Aucun travail de

mise en forme pour le respect des conventions d'appel du langage de script n'est nécessaire, comme c'est souvent le cas pour embarquer du *C/C++*. De nombreux *paquetages* sont proposés en standard dans *oRis*. Ceux-ci concernent dans l'ensemble des services tout à fait classiques mais néanmoins indispensable pour que l'outil soit réellement utilisable dans des conditions variées. Les thèmes abordés par ceux-ci sont les suivants : les conversions de données, les mathématiques, les sémaphores d'exclusion mutuelle (voir la section 6), les liens réflexes (voir la section 4.2), les conteneurs génériques, les composants d'interface graphique, le tracé de courbes, les objets situés dans un plan ou dans l'espace (voir la section 3.1), les outils de *réflexion*, les inspecteurs graphiques, la communication par fichiers, *socket*, *IPC*, le contrôle de session à distance, l'interfaçage avec *Java* et *Prolog*, les contrôleurs flous.



Figure 2. *L'environnement graphique d'oRis*

Un autre aspect d'*oRis* concerne l'environnement graphique qu'il propose à l'utilisateur. Par défaut le lancement d'*oRis* fait apparaître une console graphique permettant le chargement des applications, leur lancement, leur suspension ... Comme le montre la figure 2, la console (en haut à gauche) propose un éditeur de texte représentant un des nombreux moyens d'effectuer des interventions *en ligne*. Les autres fenêtres représentées ici sont de simples objets de l'application : une vue sur le monde tridimensionnel, un traceur de courbes et des inspecteurs. Ces objets, et bien d'autres, sont directement accessibles depuis la console mais peuvent bien entendu être créés et contrôlés par programme.

2.3. *Des objets et des agents*

Un SMA n'est, bien sûr, pas uniquement composé d'agents. Ainsi l'environnement des agents est composé d'*objets passifs* (qui ne font rien tant que l'on interagit pas avec eux) et d'*objets actifs* (qui agissent sans être sollicités et qui génèrent des événements). Les agents créent, modifient, détruisent aussi des objets tels que des messages et des connaissances qui – bien que pouvant à un certain niveau être conçus comme des agents – sont bien implémentés au niveau terminal de la récursion comme des objets. De même, les interactions entre entités ne se font pas toutes ni sous formes d'*actes de langage* interprétés de manière asynchrone par leur destinataire, ni sous

forme de trace dans l'environnement. Les agents perçoivent les caractéristiques des objets ou des autres agents (e.g. la position géographique d'un agent, le contenu d'un message...) ce qui se traduit par la lecture d'un attribut de l'objet considéré. Symétriquement, la modification de l'environnement revient à modifier un attribut. De plus, le comportement d'un agent peut résulter d'une action non intentionnelle. Imaginons deux personnages qui « discutent » dans la rue et qui se font « bousculer » par des passants : ils décident chacun de leur propre interprétation des messages qu'ils s'échangent, mais ils subissent les collisions des passants sans que leurs intentions ou une quelconque décision de leur part ne soient impliquées (« lorsqu'on se fait bousculer, on n'a pas le choix ! »).

Par conséquent, dans une même application, il doit être possible d'utiliser des entités de différentes natures :

— des *objets* et des *objets actifs* (sémantique définie dans *UML* [RUM 99]) ;

— des *acteurs* tel que le propose [HEW 73] : un acteur est une entité active qui joue un rôle en donnant la réplique conformément à un script ; son comportement s'exprime par des envois de messages.

— ou des *agents* que l'on peut définir comme des entités autonomes (il n'y a pas de script *a contrario* du modèle d'acteur) et dont le comportement repose sur la perception de l'environnement et la réalisation d'actions, cette dernière étant, éventuellement, guidée par des buts (propriété de *pro-activité*).

Fondamentalement, l'implémentation de ces différents types d'entité repose sur la notion d'*objet* (instance d'une classe), la concurrence et l'autonomie étant fondée, à ce niveau, sur la notion d'*objet actif* (i.e. un objet ayant ses propres flots d'exécution). La pro-activité des *agents* repose sur les notions de buts, de connaissances, de plans d'actions et sur un mécanisme d'inférence. La programmation logico-déductive correspond à ce besoin ; avec *oRis*, ces éléments peuvent être implémentés en *Prolog* avec lequel il est interfacé.

3. Des agents situés dans un environnement

Que ce soit parce que le système simulé possède lui-même une composante géométrique ou que ce soit une métaphore qui améliore l'intelligibilité du modèle, la simulation multi-agents peut nécessiter de recourir à des agents situés (localisables et détectables) dans un environnement géométrique, généralement de dimension deux ou trois. L'introduction de dimensions « physiques » à l'environnement, enrichit la sémantique de notions fondamentales des SMA telles que :

— la *localité* : elle n'est plus seulement *logique* (i.e. conditionnée par l'organisation du SMA) mais elle correspond à une proximité spatiale (distance entre deux objets) ou topologique (un objet est dans un autre, ce dernier étant convexe, ou est attaché à un autre) ;

— la *perception* : elle ne repose plus uniquement sur l'identification d'une instance et de ces attributs (sémantique applicative), mais aussi sur la localité spatiale ou la

détection des contours et les agents doivent donc disposer de « capteurs » appropriés ayant un certain *champ de perception*.

— les *interactions* : on peut alors parler de notions telles que les collisions, l'attachement d'objets...

oRis offre des environnements bidimensionnels et tridimensionnels dans lequel les objets sont définis par leur position, leur orientation et leur forme géométrique. Les fonctionnalités de tels objets sont très similaires. Qu'ils s'agisse d'`Object2d` ou d'`Object3d`, *oRis* offre bien évidemment des outils permettant à l'utilisateur de s'immerger dans le SMA : l'utilisateur, comme tout agent, a une vision locale de l'environnement, peut être perçu par les autres agents, peut interagir « physiquement » avec eux en provoquant (avec un dispositif de pointage ou le clavier) des événements perceptibles par les agents. Le temps est aussi une variable de l'environnement qui conditionne le comportement des agents et qui permet de *situer* leurs actions et leurs interactions. *oRis* offre à ce sujet des fonctionnalités de base.

3.1. *Environnements bidimensionnel et tridimensionnel*

La figure 3 illustre le repérage des objets dans le plan et la notion de champ de perception. Les méthodes de perception fondées sur ce principe assimilent les objets situés à des points (origine de leur repère local). Le champ de perception est constitué d'un angle d'ouverture et d'une portée. Dans cette zone, les objets repérés sont localisés en coordonnées polaires dans le repère local de l'entité qui perçoit afin d'exprimer facilement une réaction aux perceptions locales (« sur ma gauche », « aller tout droit »). Ces méthodes de perception permettent de choisir le type des objets à percevoir et admettent deux variantes : détection de l'objet le plus proche ou de l'ensemble des objets visibles. Une méthode de *lancer de rayons* permet de détecter finement les contours de ces objets dont la forme est choisie parmi un ensemble de primitives. Pour réaliser des attachements, les objets peuvent être connectés entre eux de manière hiérarchique de telle sorte que le déplacement d'un objet entraîne le déplacement de tout ceux qui lui sont connectés.

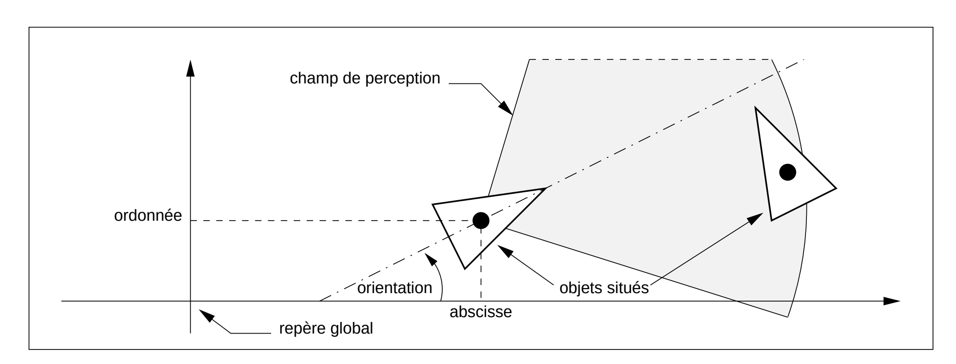


Figure 3. *Localisation et perception en 2D*

Pour aller au-delà des problèmes plans, *oRis* propose un environnement géométrique tridimensionnel. Il reprend les mêmes principes que ceux de l'environnement bidimensionnel en y ajoutant cependant les paramètres géométriques supplémentaire (six degrés de liberté). La représentation géométrique (en *OpenGL*) d'une entité est décrite sous la forme d'un ensemble de points définissant des faces, pouvant être colorées ou texturées. Des volumes élémentaires et des primitives de transformation sont également disponibles ; la construction d'une forme complexe se fait alors par accumulation d'autres formes.

3.2. *Le temps*

Les différents modes de gestion du temps classiquement utilisés en simulation [FUJ 98] reposent au plus bas niveau, soit sur un temps logique (événementiel), soit sur un temps physique. *oRis* fournit à ce titre deux moyens de mesurer le temps. La fonction `getTime()` permet de mesurer des durées physiques en millisecondes pour fournir par exemple une impression de temps-réel à l'utilisateur. La fonction `getClock()` quant à elle indique le nombre de *cycles d'exécution* écoulés, ce qui peut être assimilé à un temps logique. Notons que le sens que l'on peut donner à cette valeur dépend du type de multi-tâches utilisé, et d'une manière plus générale du contexte applicatif de la simulation.

4. Les interactions entre agents

Les interactions entre un agent et son environnement se font par la manipulation des caractéristiques structurelles des objets qui le composent. Cela se traduit par la lecture ou la modification des attributs des objets cibles, cette dernière pouvant se faire par invocation d'une méthode (e.g. appel de la méthode `move()` d'un objet avec lequel un agent situé entre en collision). Ces interactions peuvent aussi se traduire par l'instantiation d'objets ou leur destruction (on fait appel aux opérateurs `new` et `delete`). Dans tous les cas, la réaction des objets de l'environnement est impérative et synchrone.

Comme cela a été souligné dans la section 2.3, les interactions entre agents peuvent être de différentes natures. Elles peuvent être réactives (exemple de la collision) ou par échanges de messages. À cet effet, le langage *oRis* offre quatre solutions qui peuvent être utilisées simultanément dans l'implémentation d'un agent. Ces services sont assurés par la classe `Object`.

4.1. *Les appels synchrones*

Bien que le modèle objet utilise le concept d'*envoi de messages* pour décrire la communication entre objets, dans la pratique les langages implémentent un mécanisme d'appel synchrone de méthodes. L'invocation d'une méthode sur un objet ne

crée pas un nouveau flot d'exécution chez l'instance concernée mais détourne simplement l'activité de l'objet appelant vers un traitement qui manipule les données de l'objet désigné. Ceci revient à considérer que c'est l'objet appelant qui effectue tous les calculs ; l'objet concerné ne fournit que les données et leur « mode d'emploi » mais ne participe pas activement au traitement.

La programmation impérative par appel synchrone présente l'avantage d'être efficace à l'exécution et d'assurer la séquentialité des actions : le code qui suit l'invocation peut compter sur le fait que le traitement demandé a bien eu lieu puisque c'est le même flot d'exécution qui l'a effectué. Certains comportements d'agent, tel que la perception des autres, peuvent être programmés efficacement par ce moyen. Notons aussi que l'invocation d'une méthode peut se faire dans un bloc `start{}` (cf 6.1) et donc être exécutée par un flot d'activité différent du flot appelant.

Comme cela est spécifié en *UML*, les langages *C++* ou *Java* offrent un moyen de préciser le contrôle d'accès (*publique*, *protégé* ou *privé*) aux attributs et aux méthodes des classes d'objets. Pour des raisons d'efficacité, ce contrôle est réalisé à la compilation. Cette sémantique n'est pas la plus pertinente dans le cas d'agents. En effet deux instances d'une même classe peuvent intervenir directement sur leurs parties privées mutuelles, ce qui viole le principe d'autonomie si la cible est une méthode qui ne devrait être exécutée que sous le contrôle de l'agent auquel on s'adresse. Pour apporter un élément de réponse à cette situation, *oRis* propose un moyen de restreindre l'accès aux méthodes (voire même aux fonctions). Alors que `this` désigne l'objet sur lequel est invoquée la présente méthode, le mot-clef `that` indique l'objet qui a invoqué cette méthode sur l'objet désigné par `this`. En vérifiant l'identité de `that` au début d'une méthode, il est possible de contrôler l'accès au service correspondant. Nous pouvons ainsi, de manière très simple, vérifier que certains services ne sont directement accessibles que par les entités concernées. Les vérifications peuvent bien entendu porter sur le schéma classique à base de classes mais peuvent reposer sur des modalités propres à l'application et sur des conditions variables dans le temps. Remarquons que ce moyen est équivalent au champ *émetteur* d'un message asynchrone (voir les sections 4.3.1 et 4.3.2) ce qui permet à un agent destinataire de maintenir les mêmes raisonnements quel que soit le support de la communication (appel synchrone de méthode ou envoi de message asynchrone). L'accès à `that` repose en interne sur le fait qu'il est possible en *oRis* d'inspecter la pile d'exécution du traitement en cours. Nous pouvons par exemple n'autoriser l'invocation d'une méthode que depuis quelques autres.

Bien qu'ils aient certainement un coût en terme de vitesse d'exécution, ces contrôles d'accès dynamiques ouvrent quelques perspectives concernant la programmation de systèmes multi-agents. Ceci permet d'édicter des règles d'interaction au sein de structures organisationnelles et de les rendre dynamiques. Ainsi, est-il envisageable de donner à un agent le moyen de refuser d'exécuter un service parce que le demandeur n'a pas d'autorité sur lui dans la structure organisationnelle ou parce que le service pourrait avoir été demandé, même indirectement, par un agent qui se serait montré non coopératif dans une précédente demande.

4.2. *Les liens réflexes*

oRis fournit un mécanisme de programmation événementielle, appelé *liens réflexes*, qui permet à un agent de réagir à la modification d'un attribut d'un objet, qu'il s'agisse d'un de ses attributs (état interne de l'agent, boîte aux lettres...) ou l'attribut d'un autre objet (cas d'un agent qui « surveille » un objet de l'environnement où une caractéristique perceptible d'un autre agent).

Un *lien réflexe* est un objet assigné à un attribut et qui voit certaines de ses méthodes se déclencher automatiquement à chaque fois que l'attribut en question est modifié. Ainsi, lors de la modification d'un attribut auquel un lien réflexe est associé, celui-ci est activé, ce qui déclenche automatiquement l'exécution de sa méthode `before()` juste avant que la modification ait lieu, et de sa méthode `after()` juste après. Il est possible d'associer un nombre quelconque de *liens réflexes* sur un même attribut d'une même instance.

Nous assimilons ce mécanisme à un moyen de communication (dans un contexte réactif) puisque le déclenchement des traitements signalant la modification d'un attribut peut être vu comme un moyen d'être tenu informé d'un événement.

4.3. *Communication par messages*

Conformément au modèle *objet* qui repose conceptuellement sur ce type d'interaction entre instances, ce mécanisme est défini en *oRis* dans la classe de base `Object`. Il utilise aussi les services de la classe de base `Message`.

4.3.1. *Les envois de messages en point-à-point*

Pour envoyer, un message, un agent instancie un objet d'une classe dérivée de `Message` (dont le seul attribut est la référence de l'émetteur qui est déterminé grâce au mot-clef `that` présenté en section 4.1). L'invocation de sa méthode `sendTo()` le place dans la boîte aux lettres (FIFO) du destinataire dont la référence est précisée en argument. Pour que ce dernier puisse traiter le message reçu, il doit consulter sa boîte aux lettres. Le destinataire peut être informé qu'un message y a été déposé par le déclenchement automatique d'une méthode réflexe. On peut par exemple relancer l'exécution de l'activité de lecture des messages qui aurait pu être suspendue afin d'éviter toute attente active. Le type de communication décrit ici est qualifié de point-à-point dans le sens où l'expéditeur envoie son message à un destinataire qu'il connaît et qu'il désigne explicitement. Ce procédé est asynchrone puisque l'émetteur poursuit ses activités sans savoir exactement quand son message est traité par le destinataire.

4.3.2. *La diffusion de messages*

En prolongement du modèle objet, un agent peut envoyer simultanément un message à des objets (agents) qu'il ne connaît pas et qui déclencheront à réception du message telle ou telle action inconnue de l'émetteur. Les messages utili-

sés sont en tout point identiques à ceux évoqués précédemment. La différence repose sur la manière de les expédier et de les recevoir. L'émetteur invoque cette fois la méthode `broadcast()` du message afin qu'il soit diffusé à l'ensemble des instances qui peuvent être concernées. Leur détermination se fait à l'aide de la méthode `setSensitivity()` de la classe `Object`. L'invocation de cette méthode requiert deux arguments qui désignent respectivement le type de message diffusé auquel l'instance devient sensible (le nom de la classe), et le nom de la méthode qui doit être déclenchée lors de la réception d'un tel message (e.g. `setSensitivity("WhoAreYou", "presentMySelf")`). Lorsqu'un message est diffusé, cela provoque instantanément, sur chaque objet sensible à ce type de message (classe `WhoAreYou`), l'invocation de la méthode qu'il a spécifiée (`presentMySelf`). Ce mécanisme réflexe convient bien aux messages que l'on considère comme des événements et qui nécessitent une réaction immédiate de la part des objets qui l'interceptent. Cependant, si ce n'est pas le cas, les messages reçus par diffusion peuvent aussi être placés dans la boîte aux lettres du destinataire et donc être traités comme s'ils avaient été envoyés en point-à-point ce qui homogénéise le mode de réaction de l'agent destinataire.

Un même objet peut être sensible à plusieurs types de messages et il existe un mécanisme rappelant le suivi des liens polymorphes concernant le choix des méthodes réflexes à déclencher. La sensibilité aux différents types de messages peut évoluer dans le temps afin de changer de réaction ou de devenir insensible à certains types de messages.

5. L'utilisateur : spectateur, acteur et créateur

Pour que l'utilisateur soit plus qu'un simple spectateur de l'évolution du SMA, il faut qu'il dispose d'un langage ayant des propriétés dynamiques afin que de nouvelles portions de code puissent être prises en compte à tout instant et dans des circonstances variées. Les interventions les plus simples consistent à déclencher de nouveaux traitements pour changer le déroulement naturel de l'évolution du SMA. La construction incrémentale du système nécessite de pouvoir compléter l'application en cours de fonctionnement en y ajoutant de nouvelles notions, et notamment de nouvelles classes. Les modifications peuvent aussi ne concerner que des instances isolées. Toutes ces modifications *en ligne* permettent à l'utilisateur de considérer les modèles qu'il utilise comme étant eux-mêmes des paramètres de l'application.

Pour que ces manœuvres soient aisées, la grammaire du langage doit permettre de préciser très facilement si une intervention concerne le contexte global de l'application, une classe ou une instance. Ces informations contextuelles permettent d'utiliser un interpréteur ayant un point d'entrée unique pouvant être alimenté par une trame dont l'origine importe peu (réseau, zone de saisie, fichier, génération automatique...). Ainsi, l'expression de ces interventions n'impose pas à l'utilisateur l'usage d'un outil particulier (un inspecteur graphique par exemple), même si de tels outils sont disponibles. Au contraire, cela autorise l'application réalisée à fournir un accès aux fonc-

tionnalités dynamiques sous la forme qui semble la mieux adaptée au contexte applicatif abordé.

Notre objectif consistant à permettre d'apporter des modifications *en ligne* est motivé par le fait que dans une *réalité virtuelle* nous estimons que « la vie doit continuer malgré tout ». Par là, nous entendons que, quoi qu'il puisse se produire, et quoi qu'un utilisateur ait pu provoquer, le modèle doit continuer à s'exécuter même si des erreurs se produisent. Il est donc nécessaire de réduire au maximum le nombre d'erreurs potentielles dans l'application. Ce point justifie pleinement le choix d'un typage fort. En effet, si les contrôles de type ont lieu dès que le code est introduit, des incohérences peuvent être détectées, ce qui permet de rejeter le code incriminé avant qu'il ne provoque des erreurs effectives dans l'application. De nombreuses autres erreurs peuvent survenir à l'exécution et elles ne doivent en aucun cas provoquer l'arrêt de l'application. Pour éviter d'être obligé d'interrompre l'application afin de procéder à son « déverminage », nous préférons n'interrompre que l'activité dans laquelle s'est produite l'erreur. Ainsi les autres traitements continuent à s'exécuter en parallèle pendant qu'un utilisateur intervient sur le traitement incriminé.

Pour faciliter l'interaction de l'utilisateur avec l'application et ses constituants, *oRis* propose quelques mécanismes simples et directement utilisables qui sont apparentés à de l'*introspection*. D'autres services de *réflexion* plus classiques font l'objet d'un paquetage particulier. Une première facilité concerne le nommage des objets et l'expression des *références*. Chaque instance créée reçoit automatiquement un nom, lisible par l'utilisateur, qui le désigne de manière unique. Il est constitué du nom de la classe de l'objet créé, suivi d'un point et d'un entier servant de discriminant entre les instances d'une même classe (`MyClass.5` par exemple). Ce nom n'est pas qu'une simple fonctionnalité utilitaire mais fait partie intégrante des conventions lexicales du langage ; il s'agit d'une constante de type *référence*.

Un ensemble de services permettent de connaître l'ensemble des classes existantes ou des instances d'une classe particulière. Il est aussi possible de demander sa classe à un objet, ou encore de vérifier s'il est instance d'une classe particulière ou d'une classe dérivée.

6. Le contrôle de l'ordonnancement des agents

Le contrôle du procédé d'ordonnancement des comportements des agents – implémentés en *oRis* sous la forme d'*objets actifs* – est un point très important de notre réalisation. L'expérience montre qu'en règle générale peu de choses sont annoncées et garanties par les services multi-tâches proposés dans les divers environnements de programmation. Ce manque d'information laisse toujours planer un certain doute quant à l'équité de ces procédés et aux risques de biais qu'ils pourraient induire. Nous présentons ici le procédé d'ordonnancement retenu dans *oRis* afin que l'utilisateur sache exactement à quoi il peut s'attendre lorsqu'il décide de faire évoluer des agents en parallèle.

6.1. *Les flots d'exécution*

Le premier sujet à aborder lorsqu'on parle de multi-tâches concerne la détermination de la nature des tâches que l'on souhaite faire s'exécuter en parallèle. Bien qu'elles soient gérées de la même façon en interne, le langage *oRis* propose trois manières d'exprimer ces tâches que nous désignons par le terme *flot d'exécution*.

Un objet actif est caractérisé dans *oRis* par le fait qu'il dispose d'une méthode `main()` représentant le point d'entrée du comportement de l'instance concernée, puisque nous estimons que le rôle principal d'une telle instance consiste à exécuter cette méthode particulière. Lorsqu'une instance dotée d'une méthode `main()` est créée, cette méthode est immédiatement prête à s'exécuter. Quand la fin de la méthode est atteinte, elle est automatiquement relancée à son début. C'est donc un moyen simple d'implémenter le comportement autonome d'un agent. Une simulation multi-agents en *oRis* consiste à instancier de tels agents et à les laisser « vivre ».

Un autre moyen permettant d'initier un nouveau traitement en parallèle des objets actifs consiste à dédoubler le flot d'exécution grâce à la primitive `start`, comme illustré sur la figure 4. Ce procédé, consistant à générer plusieurs flots d'exécution à partir d'un seul, sert principalement à attribuer plusieurs activités à un même agent. Il est effectivement envisageable que le comportement principal d'un agent (sa méthode `main()`) génère d'autres activités annexes. Il semble raisonnable qu'un agent puisse par exemple se déplacer tout en communiquant avec d'autres.

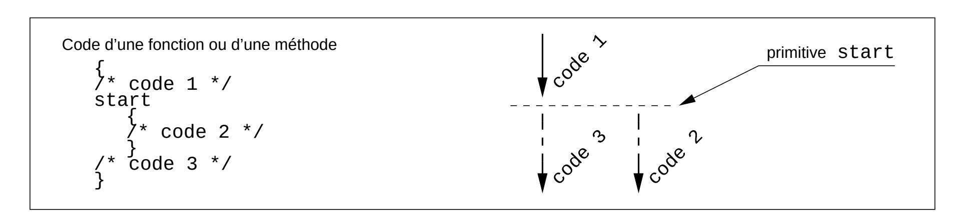


Figure 4. *Dédoublement du flot d'exécution*

Les deux types d'activités précédentes sont clairement destinées à l'écriture des comportements des agents. La troisième forme que nous présentons ici est plus directement destinée aux interventions de l'utilisateur mais, est toutefois applicable en de nombreuses circonstances. Nous réutilisons le mot-clef `start`, mais cette fois il prend place à l'extérieur de tout bloc de code. Il permet de décrire un bloc de code dont l'exécution démarre juste après son analyse par l'interpréteur. Un exemple d'utilisation d'un tel bloc de code à déjà été donné sur la figure 1. Il permet notamment de faire apparaître de nouvelles instances ou d'initier des traitements qui s'exécutent en parallèle de toutes les activités de l'application. Un tel bloc `start` peut être considéré comme une fonction anonyme, c'est-à-dire qu'il héberge des variables locales et que le code qu'il contient ne concerne pas une instance en particulier. Il est toutefois possible de le considérer comme une méthode anonyme en le faisant précéder du nom d'une instance ; le code exécuté dans le bloc `start` concerne cet

objet particulier comme s'il s'agissait d'une de ses méthodes. Une telle possibilité, lorsqu'elle est alliée aux propriétés dynamiques du langage, permet à l'utilisateur de déclencher un traitement en cours d'application en se faisant passer pour une instance particulière ; cela revient à considérer cet objet comme un *avatar* de l'utilisateur.

Quelle que soit la manière utilisée, parmi les trois présentées ici, pour créer des flots d'exécution, ceux-ci sont gérés de la même façon par l'ordonnanceur. Il sont tous désignés par un identifiant entier et peuvent être suspendus, relancés ou détruits. Un soin particulier a été apporté au traitement des erreurs pouvant survenir en cours d'exécution. Puisqu'il s'agit par essence même de circonstances imprévues, tels une division par zéro ou l'accès à une instance ayant été détruite, nous n'avons pas cherché à proposer un mécanisme de rattrapage d'erreurs « prévues » de type `try/catch` qui n'a effectivement pas vocation à corriger les erreurs mais qui permet d'en signaler clairement l'occurrence. Dans *oRis*, une erreur lors de l'exécution d'une activité provoque la destruction de ce flot et de lui seul (ainsi que l'affichage du message d'erreur et de la pile d'exécution). Cette solution consistant à n'arrêter que l'activité incriminée dans la génération d'une erreur permet d'envisager un univers qui continue à s'exécuter même quand des erreurs se produisent localement.

6.2. *Le procédé d'activation*

6.2.1. *L'architecture retenue*

L'ordonnanceur *oRis* maintient un ensemble de flots d'exécution. La figure 5 présente très sommairement les structures de données utilisées pour la gestion de ces traitements parallèles. Chaque flot d'exécution est représenté par une structure de données qui, en plus de l'identifiant de l'activité, contient une pile de contextes et une pile de valeurs temporaires. La pile de contextes sert à matérialiser les appels de fonctions et de méthodes imbriqués. Les modules exécutables sont représentés par une séquence de micro-instructions. Celles-ci étant atomiques, les commutations d'activité entre les différents flots ne peuvent avoir lieu qu'entre l'exécution de deux micro-instructions. La pile de données temporaires est commune à tous les contextes du flot d'exécution et permet notamment d'empiler les paramètres d'un module exécutable avant son appel, et de récupérer le résultat empilé lors de la terminaison du module appelé.

6.2.2. *Trois modes de multi-tâches*

Nous savons maintenant que les différents flots d'exécution peuvent être interrompus entre deux de leurs micro-instructions. Nous abordons alors la description du facteur provoquant l'interruption d'un flot au profit d'un autre. Ce point peut être contrôlé librement par l'utilisateur dans son programme.

Une première possibilité consiste à placer l'ordonnanceur en mode coopératif. Dans ces conditions, le changement d'activité n'a pas lieu spontanément mais repose exclusivement sur des consignes explicites inscrites dans le code à exécuter (fonction

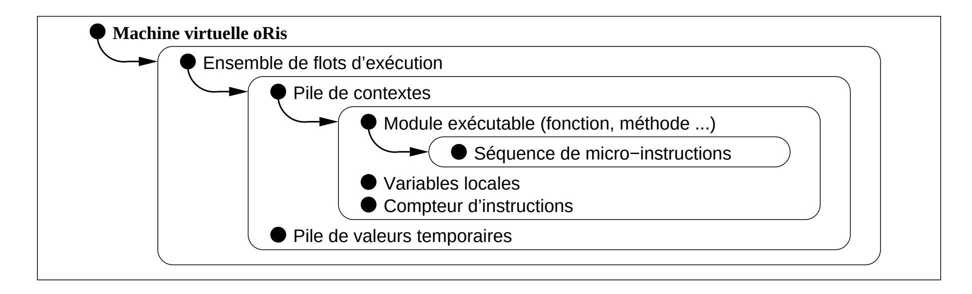


Figure 5. *Structure de la machine virtuelle oRis*

`yield()`, fin de `main()`, attente de ressource). L'ordonnanceur peut alors interrompre le traitement courant pour en faire progresser un autre. Une activité ayant laissé les autres s'exécuter reprend ses traitements lorsque toutes les autres en ont fait de même. Ce type de multi-tâches suppose que toutes les activités de l'application « jouent le jeu », c'est-à-dire qu'elles prennent régulièrement le soin de passer la main aux autres. Si l'une d'elles monopolise l'activité de l'ordonnanceur, elle fige tout le reste de l'application.

Il est également possible de choisir un ordonnancement préemptif en précisant la période en millisecondes à laquelle une commutation doit avoir lieu. Dans ces conditions, l'écriture de comportements longs se fait sans se soucier du minimum d'entrelacement que chaque activité doit assurer. Le programmeur n'a plus besoin d'insérer des commutations explicites (`yield()`) puisqu'elles sont spontanément assurées par l'ordonnanceur.

Un contrôle plus fin des commutations est envisageable en précisant cette fois un nombre de micro-instructions comme intervalle de préemption. Les notions de parallélisme et d'entremêlement sont alors poussées à leur paroxysme lorsque ce nombre vaut `1`, ce qui justifie le choix de l'appellation « profondément parallèle ». L'intérêt de ce procédé consiste à proposer un parallélisme qui va bien au delà de ce que les deux précédents modes proposent. Bien que ces derniers proposent un découpage du temps, ils reposent néanmoins sur des portions de code qui sont exécutées en séquences plus ou moins longues. Avec ce nouveau mode d'ordonnancement, nous pouvons réduire à l'extrême la longueur de ces séquences dans l'application, ce qui tend à rapprocher son exécution d'un parallélisme réel (même si ce n'est pas le cas). La contrepartie de cette possibilité réside dans le fait que l'ordonnanceur puisse passer plus de temps à effectuer des commutations qu'à exécuter le code « utile » de l'application. Notons que ce type de multi-tâches (tout comme le mode coopératif) ne dépend d'aucune horloge *système* et n'est que l'expression d'un algorithme interne à *oRis* ; il est donc de ce fait parfaitement maîtrisé.

6.2.3. *Désignation des traitements à activer*

Le dernier point à discuter à propos du procédé d'activation concerne le choix du flot à activer après une commutation. L'introduction de priorités ne ferait que

repousser le problème pour les flots de même priorité et nous semble difficile à interpréter à propos d'entités autonomes. Il nous paraît en effet préférable d'admettre que les différentes entités n'exploitent pas le temps de la même façon plutôt que de dire que certaines « vivent plus » que d'autres[5]. Afin d'assurer un partage équitable du temps entre les différentes activités, nous introduisons la notion de *cycle d'exécution* qui apporte au système la propriété suivante : chaque flot d'exécution progresse une fois et une seule par cycle. Si, au cours d'un cycle, de nouvelles activités apparaissent, celles-ci sont exécutées au cycle suivant pour garantir la terminaison des cycles. Nous proposons deux ordres d'activation au sein d'un cycle.

La solution la plus immédiate consiste à désigner les activités dans un ordre immuable, qui introduit une relation de précédence indésirable puisque l'activité de rang $n-1$ est toujours désignée avant celle de rang n. Ainsi, si deux agents sont régulièrement en compétition pour une ressource, celui ayant son comportement au rang $n-1$ prend systématiquement la ressource au détriment de celui du rang n. Notons que cette priorité parasite ne concerne pas les activités d'un point de vue global mais seulement d'un point de vue local. En effet, au niveau microscopique, la dernière activité d'un cycle précède la première activité du cycle suivant (le changement de cycle n'est, de ce point de vue, qu'une date remarquable). Cette relation de priorité n'existe qu'entre les activités qui sont relativement proches dans l'ordre du procédé de désignation.

La simulation du parallélisme impose une exécution séquentielle des portions de code des différents flots, cette relation de priorité locale ne peut être complètement éliminée. Cependant, en introduisant un ordre aléatoire dans la désignation, nous évitons qu'elle se reproduise de manière systématique de cycle en cycle puisque l'avantage donné à une activité sur une autre lors d'un cycle est remise en cause lors d'un autre cycle. Nous conservons ainsi la notion de *cycle d'exécution* qui nous fournit un *temps logique* (cf section 3.2) commun empêchant toute dérive du temps au profit d'une activité tout en éliminant le biais introduit par les relations de précédence locales.

En plus des trois modes d'ordonnancement disponibles (coopératif, préemptif, profondément parallèle), l'utilisateur peut donc choisir entre une désignation à ordre fixe (fortement déconseillée) et une autre aléatoire permettant d'éliminer toute relation de précédence locale qui pourrait conduire à un biais. Notons que le procédé que nous avons utilisé ne repose que sur des algorithmes et des structures de données dont nous avons l'entière maîtrise et qui sont détaillées dans [HAR 00]. Il ne dépendant notamment d'aucune fonctionnalité du système sous-jacent (mis à part l'horloge du mode préemptif).

Concernant les accès concurrents à des ressources communes, *oRis* propose de très classiques sémaphores d'exclusion mutuelle. Il permettent en particulier de choisir si l'opération de dévérouillage doit avoir lieu selon l'usuelle file d'attente ou bien selon un ordre aléatoire afin d'éviter de nouvelles relations de précédence locale. Une

5. Le lièvre vit-il « plus souvent » que la tortue ou court-il simplement plus vite ?

autre solution nous est offerte pour assurer ce type de service. Il s'agit de créer des sections critiques de bas niveau (blocs `execute` semblable à `start`) qui interdisent toute commutation de l'ordonnanceur au sein de la portion de code encadrée.

Une dernière précision à propos du parallélisme concerne l'utilisation de *thread système* pour encapsuler l'invocation d'appels *système* bloquants. En effet l'ordonnanceur *oRis* est vu comme un processus unique du point de vue du système et est donc susceptible d'être suspendu dans sa globalité. Lorsqu'un traitement potentiellement bloquant doit être effectué il est lancé dans un *threads* dont l'ordonnanceur *oRis* vérifie régulièrement la terminaison.

7. La dynamicité du système multi-agents

7.1. *L'introduction de code*

Il existe en apparence de nombreux moyens d'introduire du code *oRis* en cours d'exécutions (composition d'une chaîne, fenêtre de saisie, lecture d'un fichier, d'une trâme réseau ...), mais tous convergent vers un point d'entrée unique : la fonction `parse()`. Cette fonction transmet à l'interpréteur la chaîne passée en argument, ainsi l'intégralité du langage est disponible en ligne. En effet, le même procédé est utilisé pour le chargement initial et pour les interventions *en ligne*. Il est donc possible de réaliser des agents qui, disposant d'un mécanisme d'apprentissage, pourraient acquérir de nouveaux attributs et faire évoluer leur comportement de manière autonome.

7.2. *Le déclenchement de traitements*

Le premier type d'intervention dans un programme en cours d'exécution consiste à déclencher de nouveaux traitements. Ceux-ci permettent notamment de créer de nouveaux agents, d'inspecter le modèle, d'interagir avec les agents ou de les détruire. De telles lignes de code trouvent leur place dans un bloc anonyme `start` ou `execute` (voir la section 6). Comme nous l'avons vu précédemment, ces blocs de code sont pris en charge par l'ordonnanceur dès qu'ils sont analysés par l'interpréteur. Le bloc `start` initie un traitement qui continue à s'exécuter en parallèle des autres activités alors que le bloc `execute` est exécuté de manière instantanée et ininterrompue. L'utilisateur peut déclencher des traitements en se faisant passer pour un objet de l'application en précisant le nom de cet objet devant le bloc `start` ou `execute`. L'objet devient alors un *avatar* de l'utilisateur.

7.3. *Les interventions sur les fonctions*

L'introduction de code source nous permet, à tout moment, d'ajouter de nouvelles fonctions (celles qui n'existaient pas sont crées) et de modifier les fonctions existantes (celles qui existaient sont remplacées). La déclaration d'une fonction (son prototype)

peut même remplacer sa définition ; dans ce cas l'appel à la fonction est interdit jusqu'à ce qu'une nouvelle définition soit introduite. Ces modifications concernent également le code compilé, en autorisant le choix entre plusieurs implémentations fournies en *C++* ou en retournant à une définition en *oRis*.

7.4. *Les interventions sur les classes*

De la même façon qu'il est possible d'intervenir dynamiquement sur les fonctions, *oRis* permet d'ajouter, de compléter et de modifier des classes en cours d'exécution. L'ajout d'une classe peut intervenir à tout moment en la définissant lors d'un appel à la fonction `parse()`. Lorsque l'interpréteur rencontre une définition de classe, il peut s'agir d'une toute nouvelle classe, auquel cas elle est créée, ou bien d'une classe existante, auquel cas elle est complétée ou modifiée. Il est ainsi possible d'ajouter des attributs, des méthodes et de récrire des méthodes existantes. L'ajout de méthode est alors très similaire à l'ajout de fonctions : ce qui n'existait pas existe désormais et ce qui existait est remplacé. D'une manière générale, les remarques concernant les multiples définitions et déclarations des fonctions s'appliquent aux méthodes. La situation est cependant un peu plus délicate car les effets des nouvelles définitions se combinent avec les effets des surdéfinitions. Une modification qui concerne une méthode d'une classe particulière n'influe sur les classes dérivées que si ces dernières ne surdéfinissent pas la méthode en question. En revanche, lorsqu'il n'y a pas de surdéfinition, la modification met à jour instantanément toute la hiérarchie de classes dérivées ainsi que les instances. De la même façon, l'ajout d'un attribut est répercuté sur les classes dérivées et les instances.

7.5. *Les interventions sur les instances*

Une opération encore plus fine consiste à spécialiser dynamiquement le comportement d'une instance ; la classe représente alors le comportement par défaut de l'instance. Nous réutilisons dans ce cas exactement les mêmes principes que pour les fonctions et les classes, à savoir : ce qui n'existe pas déjà est ajouté et ce qui existe est remplacé. La particularité de ces interventions vient du fait qu'elles s'appliquent à des entités déjà créées. Il s'agit d'une distinction très marquante avec les classes anonymes de *Java* qui sont créées et compilées bien avant que les instances n'existent. Nous apportons ici un moyen d'ajuster le comportement d'une instance alors qu'elle est en situation pour lui permettre d'avoir une meilleure conduite dans le cas rencontré.

La figure 6 donne un exemple d'utilisation de la fonction `parse()` pour modifier le comportement d'une instance. La composition d'une chaîne qui concerne une des trois instances créées permet d'exprimer l'ajout d'un attribut et d'une méthode ainsi que la surdéfinition d'une méthode existante. La distinction entre la classe et l'instance se fait naturellement par l'usage de l'opérateur de résolution de portée (`::`). De telles modifications peuvent bien entendu avoir lieu à plusieurs reprises en cours d'exécution.

```
class Example                             // Definition de la classe Example
{
void new(void) {}
void delete(void) {}
void main(void)                           // Methode main() initiale
  { println(this," is an Example !"); }
};

execute                                   // Code a executer
{
Example ex;
for(int i=0;i<3;i++) ex=new Example;      // Instancier trois Example
string program=
format("int ",ex,"::i;                    // Ajouter un attribut a `ex'
        void ",ex,"::show(void)           // Ajouter une methode a `ex'
        { print(++i,\" --> \"); }
        void ",ex,"::main(void)           // Surdefinir le main() de 'ex'
        { show(); Example::main(); }");   // pour utiliser ce qui a ete ajoute
println(program);                         // Afficher la chaine composee
parse(program);                           // Interpreter la chaine composee
}
```

```
int Example.3::i;                    // Ajouter un attribut a `ex'
        void Example.3::show(void)          // Ajouter une methode a `ex'
        { print(++i," --> "); }
        void Example.3::main(void)          // Surdefinir le main() de 'ex'
        { show(); Example::main(); }
Example.3 is an Example !
Example.2 is an Example !
Example.1 is an Example !
Example.1 is an Example !
1 --> Example.3 is an Example !
Example.2 is an Example !
Example.1 is an Example !
Example.2 is an Example !
2 --> Example.3 is an Example !
```

Figure 6. *Modification d'une instance*

Les fonctionnalités dynamiques exposées ici ne demandent aucune technique ou astuce de programmation particulière dans le sens où il n'y a aucune différence entre la forme du code que l'on rédige par avance et celle du code que l'on introduit dynamiquement. Les règles de réécriture sont les mêmes dans tous les cas. Cela permet notamment d'effectuer la mise au point d'un comportement sur une instance particulière afin de généraliser par la suite ce comportement à toute une classe en changeant simplement la portée du code introduit. La mise à disposition de constantes de type *référence sur un objet* permet de facilement agir sur une instance quelconque. En effet, si cette forme lexicale n'existait pas, il faudrait impérativement mémoriser une référence sur chaque instance afin de pouvoir s'y adresser au moment jugé opportun. En *oRis*, l'utilisateur n'a pas à se soucier de ce genre de détail ; si par un moyen quelconque (inspecteur graphique, pointage dans une fenêtre ...) nous faisons apparaître le nom d'un objet, nous pouvons le réutiliser pour lui faire exécuter des traitements ou pour le modifier.

8. Applications

L'ensemble de l'environnement et du langage *oRis* représente plus de cent mille lignes de code source principalement écrites en *C++* mais aussi en *Flex++* & *Bison++* et en *oRis* lui-même. Celui-ci est tout à fait opérationnel et stable et est d'ailleurs utilisés dans de nombreux projets. Il peut être utilisé librement en le télé-

chargeant depuis `www.enib.fr/~harrouet/oris.html`. Une documentation, des exemples, des supports de cours et le mémoire de thèse de Fabrice Harrouet sont également disponibles depuis cette page.

8.1. *Champs d'application*

oRis est utilisé comme outil pédagogique dans certains enseignements dispensés à l'École Nationale d'Ingénieurs de Brest (ENIB) : programmation par objets concurrents, systèmes multi-agents, réalité virtuelle distribuée, commande adaptative. Il est aussi utilisés par d'autres équipes pédagogiques : Écoles militaires de Saint-Cyr Coëtquidan, ENSI-Bourges, ENST-Bretagne, IFSIC, IRIT, IUT de Bordeaux, Université de Caen.

oRis est la plate-forme sur laquelle les travaux de recherche du Laboratoire d'Informatique Industrielle (LI2) sont menés, ce qui se traduit par le développement de paquetages de classes. Parmi ceux-ci, citons des paquetages permettant : la coordination d'actions selon le *Contract Net Protocol*, la distribution des agents [ROD 99], la communication entre agents en utilisant *KQML* [NÉD 00], l'utilisation des cartes cognitives flous [PAR 01], la déclaration de plans d'actions collectifs en utilisant une extension exécutable de la logique temporelle de Allen [DEL 00], la définition de comportements d'agents sous forme de tendances [FAV 01]. Il a aussi servi en traitement d'images [BAL 97b], en simulation médicale [BAL 97a], en simulation de systèmes manufacturier [CHE 99b], et pour le développement d'une plate-forme de formation pour la sécurité civile [QUE 01].

D'autres équipes de recherche ont aussi utilisé *oRis* pour leurs travaux : le CREC pour la simulation de champs de bataille, de conflits, de guerre électronique, le GREYC pour la simulation de réseaux informatiques, les équipes SMAC et GRIC de l'IRIT, l'UMR CNRS 6553 d'Éco-biologie...

8.2. *La plate-forme ARéVi*

oRis a été utilisé pour le développement de la plate-forme *ARéVi* (**A**telier de **Ré**alité **Vi**rtuelle) [REI 98]. Son noyau n'est autre qu'*oRis*, et donc toutes les potentialités décrites dans cet article sont disponibles ; il est étendu par du code *C++* offrant des fonctionnalités propres à la réalité virtuelle. Cette plate-forme offre un rendu graphique complètement indépendant de celui qui est proposé par *oRis*. Les objets graphiques sont chargés directement à partir de fichiers au format *VRML2* (il est possible de définir des animations et de gérer les niveaux de détails). Des éléments graphiques tels que des textures transparentes ou animés, des sources lumineuses, des *lens flares* (reflets du soleil sur une lentille) et des systèmes de particules (jets d'eau) sont disponibles. *ARéVi* introduit aussi des notions de cinématique (vitesses et accélérations linénaires et angulaires), ce qui enrichit les possibilités d'expression de comportements des agents dans un environnement tridimensionnel. Pour ce qui est du domaine

sonore, *ARéVi* propose une sonorisation tridimensionnelle et des fonctionnalités de synthèse et de reconnaissance vocale. Cette plate-forme gère des périphériques variés tels un gant de donnée, une manette de commande, un volant, des capteurs de localisation et un casque de vision qui étendent les possibilités d'immersion des utilisateurs dans le SMA.

9. Conclusion

Notre travail a abouti à la réalisation d'une plate-forme opérationnelle reposant sur la programmation par objets actifs concurrents. Elle fournit les services fondamentaux pour la simulation multi-agents interactive, c'est-à-dire lorsque l'utilisateur est « dans la boucle » du processus de simulation. En référence à l'approche « voyelles » [DEM 95], la lettre *U* est une facette peu prise en compte dans les plates-formes de simulation. Le *principe de substitution* entre agents et utilisateurs est à nos yeux fondamental ; il pourrait être évaluer par une sorte de « test de Turing » : l'utilisateur perçoit les actions d'autrui sans savoir s'il s'agit d'un agent ou d'un autre utilisateur et les agents réagissent à ses actions comme s'il s'agissait d'un autre agent. À tout moment, l'utilisateur peut se substituer à un agent en déléguant éventuellement une partie de son comportement à son avatar : par exemple, il prend en charge la perception et la délibération et délègue la réalisation des actions, la gestion des connaissances et la communication étant partagées. Cette substitution doit évidemment pouvoir être réversible. Du fait de cette expérience, l'utilisateur peut avoir besoin de modifier le code de l'agent auquel il s'est temporairement substitué. Pour cela, les propriétés de dynamicité du langage d'implémentation, la granularité instance, l'introspection et la réflexion du code, la maîtrise du contexte d'exécution sont des propriétés indispensables. La notion de *temps réel de simulation* [FUJ 98] est également fondamentale en simulation interactive. Bien que la plate-forme *Swarm* [SDG] offre des services intéressants concernant la gestion du temps, il n'est pas possible d'intégrer le fait que le temps de réaction de l'utilisateur est imprévisible et que, pendant ce temps-là, son environnement continue à évoluer.

Le deuxième point de notre contribution concerne le soin qui a été apporté à l'équité du procédé d'activation des objets actifs, donc des agents. L'intérêt de l'approche *individu-centré* n'est plus à démontrer, mais les résultats dépendant alors de l'exécution de modèles et non de la résolution d'équations, leur validité est tributaire de la justesse du simulateur (et bien sûr de la pertinence du modèle !). Or, le contrôle de l'ordonnancement est un point très sensible et difficile à maîtriser par un utilisateur, certes expert du domaine sur lequel porte la simulation mais pas nécessairement expert en programmation par objets concurrents. Il nous est donc apparu indispensable que ce mécanisme soit au cœur de la machine virtuelle *oRis* et ne soit en aucune façon laissé à la charge de l'utilisateur, comme c'est le cas avec *Madkit* (*in* [BOI]).

oRis ayant une vocation généraliste, aucun modèle d'agent n'est imposé. Pour la même raison, la facette organisationnelle, que l'on ne trouve pas dans tous les SMA

(par exemple dans les SMA auto-adaptatifs à agents réactifs avec communications médiatisées par l'environnement – typiquement les *ant systems* – [BON 99]), est très dépendante du modèle et n'entre donc pas dans le champs couvert par un langage d'implémentation.

Parmi les évolutions possibles de notre plate-forme, nous envisageons de compléter la couche bas niveau que constitue notre ordonnanceur par des mécanismes de nature plus événementielle et plus orientée sur la notion de temps physique. L'approche réactive [BOU 96] et les travaux de l'équipe *SIAMES* de l'*IRISA* [ARN 97] proposent des solutions intéressantes en ce sens. Une autre évolution de nos travaux tend vers la réalisation d'un environnement de réalité virtuelle distribué ayant les propriétés dynamiques d'*oRis*. Un tel outil permettra la modélisation interactive *et* coopérative qui nécessite une ininterruptibilité de l'exécution pour respecter les contraintes d'ordre temporel, « la vie continue malgré tout », et les contraintes d'ordre social, « on n'est pas seul à subir les conséquences de ses actes ». Des réflexions tant conceptuelles que techniques dans le domaine de la simulation multi-agents interactive distribuée, dont les exigences diffèrent de celles de la résolution distribuée de problème, sont à ce titre nécessaires.

10. Bibliographie

[ARN 97] ARNALDI B., DONIKIAN S., CHAUFFAUT A., COZOT R. et THOMAS G., « Real-time simulation platform for dynamic systems ». *IROS'97, Grenoble*, p. 32–41, 1997.

[BAL 97a] BALLET P., HARROUET F. et TISSEAU J., « A multi-agent system to model an human secondary immune response ». *SMC'97, Orlando (USA)*, vol. 1, p. 357–362, octobre 1997.

[BAL 97b] BALLET P., RODIN V. et TISSEAU J., « A multiagent system for detecting concentric strias ». *SPIE'97, Application of digital image processing, San Diego (USA)*, vol. 3164, p. 659–666, août 1997.

[BOI] BOISSIER O., GUESSOUM Z. et OCCELLO M., Plates-formes de développement de systèmes multi-agents. Bulletin de l'AFIA numéro 39.

[BON 99] BONABEAU E., DORIGO M. et THERAULAZ G., « L'intelligence en essaim ». GLEIZES M.-P. et MARCENAC P., Éditeurs., *Actes des 7es Journées Francophones d'Intelligence Artificielle et Systèmes Multi-Agents (JFIADSMA'99)*, p. 25–38, Saint-Gilles, Ile de la Réunion, 8–10 novembre 1999. Hermes Science.

[BOU 96] BOUSSINOT F., *La programmation réactive — Application aux systèmes communicants*. Masson, 1996.

[CHE 99a] CHEVAILLIER P., HARROUET F. et DE LOOR P., « Application des réseaux de Petri à la modélisation des systèmes multi-agents de contrôle ». *Journal Européen des Systèmes Automatisés (APII-JESA)*, vol. 33, n° 4, p. 413–437, mai 1999.

[CHE 99b] CHEVAILLIER P., HARROUET F., REIGNIER P. et TISSEAU J., « oRis : un environnement pour la simulation multi-agents des systèmes manufacturiers de production ». *2ème conférence francophone de Modélisation des Flux Physiques et Informationnels (MOSIM'99), Annecy*, p. 225–230, octobre 1999.

[DEG 00] DE GIACOMO G., LÉSPERANCE Y. et LEVESQUE H. J., « ConGolog, A Concurrent Programming Language based on Situation Calculus ». *Artificial Intelligence*, vol. 121, n° 1–2, p. 109–169, 2000.

[DEL 00] DE LOOR P. et CHEVAILLIER P., « Generation of Agent Interactions from Temporal Logic Specifications ». DEVILLE M. et OWENS R., Éditeurs., *16th IMACS World Congress 2000*, Lausanne, Suisse, 21-25 août 2000.

[DEM 95] DEMAZEAU Y., « From Interactions to Collective Behaviour in Agent-Based Systems ». *European Conference on Cognitive Science, Saint-Malo*, p. 117–132, 1995.

[FAV 01] FAVIER P.-A., DE LOOR P. et TISSEAU J., « Programming agent with purposes : application to autonomous shooting in virtual environments ». *International Conference on Virtual Storytelling*, Avignon, 27-28 septembre 2001.

[FIS 94] FISHER M., « A survey of Concurrent METATEM – The language and its applications ». *Temporal Logic - Proceedings of the First International Conference*, vol. 827 de *Lecture Notes in Artificial Intelligence*, p. 480–505, Heidelberg, Germany, 1994.

[FUJ 98] FUJIMOTO R. M., « Time Management in the High Level Architecture ». *Simulation*, vol. 71, n° 6, p. 388–400, décembre 1998.

[GAS 92] GASSER L. et BRIOT J.-P., « *Object-Based Concurrent Programming and Distributed Artificial Intelligence* », p. 81–107. Kluwer, 1992.

[GUE 99] GUESSOUM Z. et BRIOT J.-P. P., « From Active Objects to Autonomous Agents ». *IEEE Concurrency*, vol. 7, n° 3, p. 68–76, 1999.

[HAR 00] HARROUET F.. « oRis : s'immerger par le langage pour le prototypage d'univers virtuels à base d'entités autonomes ». Thèse de Doctorat en Informatique. Université de Bretagne Occidentale, décembre 2000.

[HEW 73] HEWITT C., BISHOP P. et STEIGER R., « A Universal Modular Actor Formalism for Artificial Intelligence ». *Third International Joint Conference on Artificial Intelligence*, 1973.

[IGL 98] IGLESIAS C. A., GARIJO M. et GONZÁLEZ J. C., « A Survey of Agent-Oriented Methodologies ». *Proceedings of th 5th International Workshop, ATAL'98, Intelligent Agents V : Agent Theories, Architectures and languages*, p. 317–330, Paris, juillet 1998.

[NÉD 00] NÉDÉLEC A., REIGNIER P. et RODIN V., « Collaborative Prototyping in Distributed Virtual Reality Using an Agent Communication Language ». *IEEE SMC'2000, Nashville (USA)*, octobre 2000.

[NWA 96] NWANA H. S., « Software Agents : An Overview ». *Knowledge Engineering Review*, vol. 11, n° 3, p. 205–244, septembre 1996.

[NWA 99] NWANA H. S., NDUMU D. T., LEE L. C. et COLLIS J. C., « ZEUS : a toolkit for buiding distributed mutliagent systems ». *Applied Artificial Intelligence*, vol. 13, n° 1–2, p. 129–185, 1999.

[PAR 99] PARUNAK H. V. D., Industrial and Practical Applications of DAI. WEISS G., Éditeur, *Multiagent Systems : a modern approach to distributed artificial intelligence*, Chapitre 9, p. 377–421. MIT Press, 1999.

[PAR 01] PARENTHOËN M., TISSEAU J., REIGNIER P. et DORY F., « Agent's perception and Charactors in virtual worlds : put Fuzzy Cognitive Maps to work ». RICHIR S., RICHARD P. et TARAVE B., Éditeurs., *Proceedings of Virtual Reality Internatial Conference, VIRC 2001*, p. 11–18, Laval, 16–18 mai 2001.

[QUE 01] QUERREC R., REIGNIER P. et CHEVAILLIER P., « Humans and autonomous agents interactions in a virtual environment for fire fighting training ». RICHIR S., RICHARD P. et TARAVE B., Éditeurs., *Proceedings of Virtual Reality Internatial Conference, VRIC 2001*, p. 57–64, Laval, 16–18 mai 2001.

[REI 98] REIGNIER P., HARROUET F., MORVAN S., TISSEAU J. et DUVAL T., « ARéVi : a virtual reality multiagent platform ». *Virtual Worlds 98, Paris*, p. 229–240, juillet 1998.

[ROD 99] RODIN V. et NÉDÉLEC A., « oRis : an agent communication language for distributed virtual environment ». *IEEE ROMAN'98, Pisa (Italy)*, p. 41–46, septembre 1999.

[RUM 99] RUMBAUGH J., JACOBSON I. et BOOCH G., *The Unified Modelling Language Reference Manual*. Object Technology series. Addison-Wesley, 1999.

[SDG] SDG. « Swarm Development Group ». `http://www.swarm.org`.

[SHO 93] SHOHAM Y., « Agent-oriented programming ». *Artificial Intelligence*, vol. 60, n° 1, p. 51–92, 1993.

[TIS 01] TISSEAU J., « Réalité virtuelle : autonomie in virtuo ». Habilitation à Diriger des Recherches, Université de Rennes-I, décembre 2001.

Article reçu le 8 février 2001
Version révisée le 12 septembre 2001
Rédacteur responsable : Zahia Guessoum

Fabrice Harrouet *est ingénieur et docteur en informatique. Il est actuellement maître de conférences à l'École Nationale d'Ingénieurs de Brest. Ses travaux de thèse ont consisté en grande partie en la conception et la réalisation de l'environnement oRis.*

Jacques Tisseau *est responsable du Laboratoire d'Informatique Industrielle de l'ENIB. Agrégé de physique, docteur en géophysique, habilité à diriger les recherches en informatique, il s'intéresse depuis dix ans à l'autonomisation des modèles numériques afin de peupler les univers virtuels pour les rendre plus crédibles.*

Patrick Reignier *est ingénieur et docteur en informatique. Il est actuellement maître de conférences de l'université Joseph Fourier à Grenoble et est rattaché au laboratoire GRAVIR où il s'interresse à la reconnaissance de scénarios pour la conception d'assistants virtuels dans les bâtiments intelligents. Il a travaillé durant cinq ans au sein du LI2 (ENIB) sur la problématique des plateformes multi-agents pour la réalité virtuelle.*

Pierre Chevaillier *est maître de conférences en informatique. Au sein du Laboratoire d'Informatique Industrielle de l'ENIB, ses travaux portent sur les modèles de systèmes multi-agents pour la réalité virtuelle.*